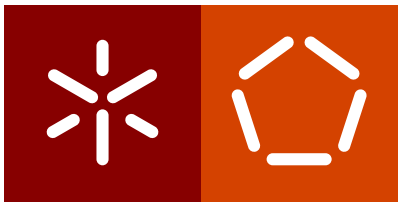
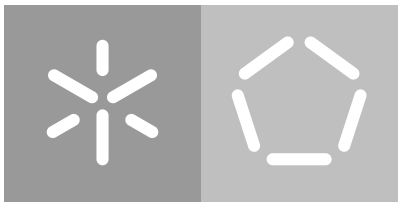


Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Nuno Castro de Macedo

Extending the BiYacc Framework with Ambiguous Grammars

October 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Nuno Castro de Macedo

Extending the BiYacc Framework with Ambiguous Grammars

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
João Saraiva
Jorge Mendes

October 2018

ABSTRACT

Contrarily to most conventional programming languages where certain symbols are used so as to create non-ambiguous grammars, most recent programming languages allow ambiguity. This results in the necessity for a generic parser that can deal with this ambiguity without loss of performance.
Currently, there is a GLR parser generator written in Haskell, integrated in the BiYacc system, developed by *Departamento de Informática (DI)*, *Universidade do Minho (UM)*, Portugal in collaboration with the National Institute of Informatics, Japan. In this thesis, this necessity for a generic parser is attacked by developing disambiguation filters for this system which improve its performance, as well as by implementing various known optimizations to this parser generator. Finally, performance tests are used to measure the results of the developed work.

RESUMO

Contrariamente às linguagens de programação mais convencionais em que certos símbolos eram utilizados por forma a criar gramáticas não ambíguas, as linguagens mais recentes permitem ambiguidade, que por sua vez cria a necessidade de um parser genérico que consiga lidar com esta ambiguidade sem grandes perdas de performance.
Atualmente, existe um gerador de parsers GLR em Haskell integrado no sistema BiYacc, desenvolvido pelo DI, UM, Portugal, em colaboração com o National Institute of Informatics, Japão. Nesta tese, são desenvolvidos filtros de desambiguidade para este sistema que aumentam a sua performance, assim como são feitas otimizações a vários níveis e se implementa um gerador de parsers usando um algoritmo GLL, que poderá trazer várias vantagens a nível de performance comparativamente com o algoritmo GLR atualmente implementado. Finalmente, são feitos testes de performance para avaliar os resultados do trabalho desenvolvido.

CONTENTS

LIST OF LISTINGS

ACRONYMS

A

AST Abstract Syntax Tree.

B

BNF Backus-Naur form.

D

DI Departamento de Informática.

G

GHC Glasgow Haskell Compiler.

GLL Generalized Left-to-Right Leftmost derivation.

GLR Generalized Left-to-Right Rightmost derivation.

L

LALR Look Ahead Left-to-Right.

LL(*) Left-to-Right, Leftmost derivation, with infinite tokens of lookahead.

U

UM Universidade do Minho.

1

INTRODUCTION

The evolution of programming languages in the 1960s was accompanied by the development of techniques for the syntactic analysis of programs. While techniques for processing text have evolved since then, the general approach has remained the same. To define and implement a new programming language, the general approach tends to be the use of context-free grammars to specify the programming language's grammar. Then, based on it, automatically generated programs (known as parsers) are produced, and these are able to syntactically recognize said programming language.

1.1 MOTIVATION

In the early ages of programming languages, it was usual to purposely include certain symbols in a language's grammar so that the generated parser for said language was more efficient.
The most obvious example is found in the C programming language: the semicolon found at the end of each instruction is a statement terminator (Perlis et al., 1981). It is used to resolve some ambiguities that could be found in the grammar. In the following example, where an excerpt of C code is listed without semicolons, there are two possible ways to interpret it.

Listing 1.1: Ambiguous C code

```
int a = b * c++
```

Listing 1.2: One possible interpretation of such code

```
int a = b; *c++;
```

Listing 1.3: Another possible interpretation of such code

```
int a = (b * c++);
```

However, most recently, for the sake of allowing the programmers to have more freedom and for them not to need to worry with the syntactic aspects of their programs, the specification of programming languages tends to allow the use of ambiguous grammars, which require powerful parsing techniques.

This report focuses on generic parsing techniques that generate a parser for any context-free grammar. Due to the ambiguity, these parsers produce various results for the same program, and disambiguation rules are used to select the desired solution, but the existing tools that provide generic parsing techniques are still limited. When these tools need to deal with ambiguity, there are only a pre-defined set of solutions, which the developer is free to use but unable to modify or elaborate on. As these tools behave as a blackbox, there is also no certainty about their correctness. They are also limiting in terms of development cycle, as they cannot be changed and tested easily and instead require a new generation and compilation process, which can be slow when the input is large.
As such, a correct and efficient implementation of such disambiguation rules is desired - this is one of the focus of this report. These rules are to be implemented as combinators, which are simple code tools, that are easy to implement but very powerful when combined with other combinators. Having these rules implemented as combinators makes them much more versatile, as well as allowing for new rules to be created, by creating new combinators. Other general improvements are also added into the current implementation of a *Generalized Left-to-Right Rightmost derivation (GLR)* parser generator studied in this work. Several performance tests are executed so as to measure the progression obtained in this work.

1.2 Document Structure

This work briefly describes some of the existing solutions for parser generation and proposes a new, alternative solution for the implementation of the disambiguation rules.
More details on existing work are explored in chapter 2, where existing ideas and solutions for parsing and disambiguation. The contribution of this work is presented in chapter 3, as well as the tools used for the implementation. Chapter 4 presents some usage examples as well as benchmarks that showcase the results of this work. Conclusions and future work are found on chapter 5, and appendices with relevant information such as grammars used

throughout the work are found in the appendices, starting from appendice A.

2

STATE OF THE ART

There are various different parsing algorithms proposed to solve the parsing problem. The earliest solution to be used was to embed the grammar in the code, where there was no separation between the grammar and the rest of the code. This was a solution that made it very difficult to change the grammar once it was coded in, since it was mixed with the code.

2.1 BNF NOTATION

The *Backus-Naur form (BNF)* notation is a notation for specifying context-free grammar, generally used for specifying the exact grammar of programming languages. It was proposed by Backus (1959), to describe the language of what became known as ALGOL 59.
In the following example, the BNF notation is used to describe the grammar of arithmetic expressions.

Listing 2.1: Grammar of arithmetic expressions

```
Exp :     Exp '+' Exp
        | Exp '-' Exp
        | Exp '*' Exp
        | Exp '/' Exp
        | '(' Exp ')'
        | int
```

This grammar is inherently recursive: an expression (Exp) can be defined as an expression (Exp), an arithmetic sign ('+') and another expression (Exp). The entirety of a grammar is expressed in this way, which is simple to understand and powerful when compared to embedding the grammar in the code.
The BNF notation is extremely interesting as it laid the foundation to having the grammar separated from the code, such that it would be easy to change the grammar without having

to change any of the remaining code.

2.2 COMMON PARSERS

While it is common for programming languages to be expressed as grammars, for example using the BNF notation, there are various ways to generate a parser given such specification. There are various alternative parsing methods, which have their own advantages and disadvantages. The first to be relevant were the most powerful in terms of compilation and execution time. As such, the grammars were changed to best fit the method: the programmer had to both focus on writing the correct grammar and adapting it to fit the parser generator. The previous example uses left-recursion, which is impossible to parse (Aho et al., 2006) for some of these algorithms. Therefore, the previous example would have to be changed so as to not have left-recursion, before it could be used in some parser generators.

One of the most well-known parser generators, Yacc, is a *Look Ahead Left-to-Right (LALR)* parser generator (Johnson, 1979): this is a relatively lightweight algorithm which was perfect for the time it was developed, that is, 1975, when it was much more necessary to restrict program runtime and memory size.

The LALR algorithm (Deremer, 1969) appeared as an algorithm which could combine the relatively small size of parse tables of the LA(0) algorithm which couldn't handle some grammars, with the power of the LA(1) algorithm which could handle said grammars but would generate a bigger parse table. This way, this algorithm combines the size of LA(0) with the power of LA(1). While there were still some grammars that could be parsed by the LA(1) algorithm but not by the LALR algorithm, it was still powerful enough for most mainstream programming languages.

Another example of a popular parser generator is ANTLR (Parr and Fisher, 2011), which is a *Left-to-Right, Leftmost derivation, with infinite tokens of lookahead (LL(*))* parser generator and whose development started in 1989. The LL(*) algorithm allows for parsing decisions to be taken by looking at the following tokens in the input stream.

While Yacc generates C code and ANTLR generates code for various programming languages, one of the most popular parser generators for Haskell is Happy (Marlow and Gil, 2001), which enables the developer to supply a file with the specification of a grammar, and in turn generates a parser, that is, a module of code that can read text according to that grammar's specifications. Happy is part of the Haskell Platform, being one of the most famous Haskell parsing tools. Due to it's rather big popularity and regular maintenance, it is a fairly well optimized tool.

Listing 2.2: Grammar of arithmetic expressions with actions

```
Exp :     Exp '+' Exp     { Plus $1 $3 }
          | Exp '-' Exp { Minus $1 $3 }
          | Exp '*' Exp { Times $1 $3 }
          | Exp '/' Exp { Div $1 $3 }
          | '(' Exp ')' { $1 }
          | int         { Int $1 }
```

In listing 2.2, the example of the grammar for arithmetic expressions is revisited. The grammar specification is similar to the BNF notation, but it also has computational load to be executed at each step written at the right-hand side, in brackets.
For the same example, after recognizing a sum between two expressions, this example would build a Haskell data type using the constructor Plus followed by the two expressions parsed. By having such a grammar specified, Happy will automatically build a Haskell module that converts the string representing the input computation, to an *Abstract Syntax Tree (AST)* with an intern representation of said computation. This is the core *modus-operandi* for any parser generator, it simplifies the developer's work by automatically generating part of the code in an efficient and useful way.

2.3 Generalized Parsing

Most parsing techniques do not deal with ambiguity properly. The input is expected to be ambiguous, and when it is not, a certain interpretation of said ambiguity is chosen so as to continue parsing. This results in efficient but not so powerful parsers, as they ignore any ambiguity problems that could arise. This is efficient as the parser is skipping some work, but at the same time sometimes impractical, as the wrong interpretation could be selected causing the parser to work in ways the developer does not want it to.
In the early days of programming languages and parsing problems in general, performance was an issue. The hardware was simply much slower than it is now, and so the priority was to optimize parsers as much as possible. As such, there was a reduced interest in generalized parsing in the past.
Recently, there has been an increasing interest in GLR parsers, which are slower than their non-generalized counterparts, due to their additional flexibility in dealing with ambiguity: when faced with an input with several different possible outputs due to ambiguity, a GLR parser (Tomita, 1985) will produce all of the outputs instead of selecting one of them. If no ambiguity is present, a GLR parser will behave just like a LR parser (Johnstone et al., 2004), which is efficient. With the constant advances in technology, the limitations that made this technique undesirable are gone, and GLR parsers are becoming more and more popular.

However, GLR is not the only generalized algorithm. The *Generalized Left-to-Right Leftmost derivation (GLL)* algorithm is also generalized, but much less explored. While there aren't any powerful GLL parser generators on the rise in the industry, it is definitely an interesting idea. The work of Afroozeh and Izmaylova (2015) describes in detail an optimized version of this algorithm, which is worst-case cubic in both time and space.

2.4 SCANNERLESS PARSING

Generally speaking, parser applications are divided in two components: the lexer and the parser. The lexer takes the input and breaks it into a list of tokens, and then the parser takes those tokens and matches them with the production rules to produce the actual parsing result.
However, there is an alternative technique, scannerless parsing. This technique consists of skipping the lexer entirely and treating each character from the input as a token, which is fed directly into the parser. Scannerless parsing removes the necessity of describing the tokens in the grammar specification, allowing the programmer to write the grammar without worrying as much with comforming with the parser specification. Besides that, scannerless parsers can be compositional, therefore allowing for two parsers to be merged without needing to change them. The downside of this technique is that it is generally less efficient performance-wise, due to a much higher number of tokens to be processed by the parser.

2.5 DISAMBIGUATION FILTERS FOR SCANNERLESS GENERALIZED PARSING

In this work, the focus is set on scannerless generalized parsers. For such, as they deal with ambiguous inputs, it is expected to get a list of outputs as a result, which represent all possible interpretations. However, not all possible interpretations are desired: depending on the situation, a developer might want to only get one or a small subset of parse trees, instead of all the possibilities.
The act of taking the ambiguous parse trees and removing the undesired is the disambiguation. Typically, such filtering is done on the parser, that is, modifying part of the parser so that the undesired interpretations cannot be produced. Some new rules for disambiguation are needed when dealing with scannerless parsing. In this section, some filters for disambiguation are presented and described, according to Fernandes et al. (2004)'s work.

2.5.1 *Priority and Associativity Filter*

The priority and associativity filters are the most commonly known out of the disambiguation filters. The priority filter specifies that certain productions have a higher priority than others, while the associativity filter specifies that an operator associates left or right. As such, extending the grammar of arithmetic expressions seen in listing 2.1, these filters could be described as shown in listing 2.3.

Listing 2.3: Grammar of arithmetic expressions, with priority and associativity filters

```
Exp :         Exp '+' Exp   %left
            | Exp '-' Exp   %left
            | Exp '*' Exp   %left
            | Exp '/' Exp   %left
            | '(' Exp ')'
            | int

{Exp : Exp '*' Exp} > {Exp : Exp '+' Exp}
```

2.5.2 *Reject Filter*

The reject filter enables the creation of keywords in the grammar. In other words, it rejects some productions from deriving into certain sequences. This is extremely useful as in most programming languages, some keywords cannot be used as variable names, and this filter allows for a clean implementation of this incompatibility. For example, in the C programming language, it shouldn't be allowed for a variable to be named "while", as that is a reserved keyword used in defining loops.
A developer could specify such filters by describing what the production should not derive into, as shown in listing 2.4.

Listing 2.4: Reject filter for "while" reserved keyword

```
String : "while" {reject}
```

2.5.3 *Follow Filter*

The follow filter solves a less obvious ambiguity that arises in scannerless parsing. Let us consider as example the grammar specified in listing 2.5, which specifies a list of values, separated by whitespaces, in which a whitespace is defined as zero or more whitespace characters such as spaces and tabs. This example can be used to define the grammar for vector declarations of Matlab, where a vector can be declared as a list of values using only whitespace as separators.

Listing 2.5: Grammar for a list of values

```
Values :     Values Ws Values
           | Value

Ws : [ \\n\\t]*
Value : [0-9]+
```

When the input is a single number with various digits, such as "37", it is ambiguous. It can be interpreted as a single "Value", or as two separated "Values", separated by exactly zero "Ws". The second interpretation is generally unwanted, and a way to fix this would be to change the definition of "Ws" to only allow for one or more whitespaces. However, when "Ws" is the default whitespace definition used throughout the grammar, this change is undesired as it would force the grammar to be changed to accomodate this change.
Hence, in this case, the follow filter (also known as longest match filter) is defined to specify that a "Value" cannot be followed by a digit. This way, "Value" must contain all the digits before an actual whitespace occurs, therefore forcing a longest match to occur. For the example input "37", the second interpretation where two separated "Values" contain each one digit would be invalid, as the first "Values" string must be followed by a non-digit character, but is followed by the "7" digit.

Listing 2.6: Follow filter for the grammar for a list of values

```
Value -/- [0-9]
```

2.5.4 *Preference Filter*

When there are several correct interpretations of a given input but some are preferred over others, a preference filter is used. It specifies which parse results should be removed when there are several correct outputs but the developer wants to select only a part of them.
This filter is the go-to filter to remove the dangling else problem. In the listing 2.7, the dangling else problem is exemplified, with the preference filter already described. The ambiguity present here can be exemplified by the input *if bool1 then if bool2 then out1 else out2*, which can be interpreted in two ways,*if bool1 then (if bool2 then out1 else out2)* or *if bool1 then (if bool2 then out1) else out2*.

Listing 2.7: Grammar exemplifying the dangling else problem, and usage of preference filter

```
Term :     "if" Bool "then" Term        %prefer
         | "if" Bool "then" Term "else" Term
```

3

PROPOSED SOLUTION

The classical approach to disambiguation revolves around having the disambiguation rules in the grammar description. However, this is not modular at all, as every small chance a developer might need to test implies generating a new parser, since the disambiguation rules are meshed into the parser during parser generation, changing the parser. This results in having a hard time trying out different combinations of rules, as the developer needs to generate a new parser for each of the modifications to be tested.
In this paper, a new approach is described. Instead of imbuing the disambiguation rules in the parser itself, they are kept separate. The parser is generated once, and it produces a possibly ambiguous result. Afterwards, the disambiguation rules are applied, removing some or all of the ambiguities, according to what the developer wants. There are several advantages and disadvantages to using this process instead of the classical approach. Since the parser that is used is unmodified, it is less efficient, as the classical approach removes parts of the parser reducing thus the number of results the parser has to output. However, while the parser itself is less efficient, the development cycle of the programmer is more efficient, as there is no need to constantly change the parser. Only the disambiguation rules are to be changed, and this can be easily done if the implementation is user-friendly. Therefore, the disambiguation rules are implemented as filter combinators, where the developer starts with basic blocks that perform very simple filtering, combining them in easy-to-understand ways to produce complex filters that perform the desired disambiguation rules.
While this solution is less efficient, it allows for a faster development cycle, and the final version of the product could use the classical approach to produce an efficient parser with disambiguation rules, after this approach was used to define which disambiguation rules should be used.

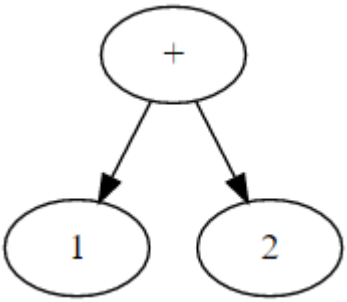


Figure 1: Syntax tree for the input "1+2".

3.1 SYNTAX TREES

A parser typically outputs the parsing result as a syntax tree. This is a tree that contains all the information from the input categorized in accordance to the grammar. As an example, according to the grammar seen in 2.2, for an input "1+2", the output can be seen in figure 1. However, for an input "1+2*3", and with no disambiguation rules defined, the parser does not understand arythmetic priority and can generate two possible outputs, as seen in figure 2. For a generalized parser, both possibilities are generated, and therefore the output is not a tree but a forest, that is, a list of trees.
Since, in this work, the disambiguation is to be performed after the parsing step, the disambiguation rules should be applied to the syntax trees, trying to find undesired patterns in them and removing them if any of these patterns are found. Such process will be described in more detail in the following sections.

3.2 HASKELL XML TOOLBOX

Syntax trees are trees in which, for each node, there is the node information and a list of children. These trees are known as Rose Trees and are already well studied in various cases. One of them is XML, for which there are several generic tools that can be used. In this work, the filter variant of the Haskell XML Toolbox (Schmidt et al., 2016) is used as a base for building combinators for filtering the syntax trees.
In this library, Rose Trees are known as *NTree*, and these trees and some tools for manipulating them are defined in the module *Data.Tree.NTree.TypeDefs*. The building blocks used in this work are defined in the module *Data.Tree.NTree.Filter*.

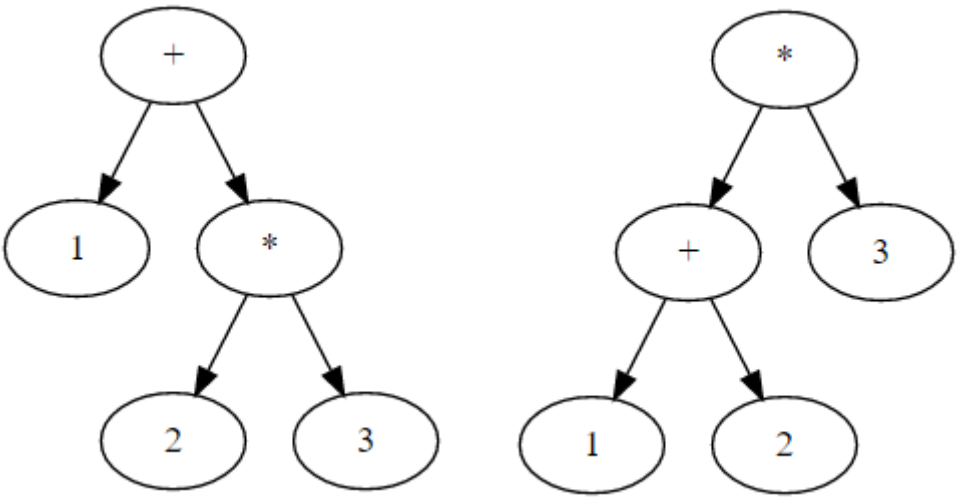


Figure 2: Syntax trees for the input "1+2*3".

3.3 HAGLR PARSER GENERATOR

The HaGLR tool (Fernandes et al., 2004) is a Haskell implementation of a GLR parser generator developed by DI/UM. It is currently extremely inefficient, as shown in Fernandes et al.' work, where AG.bib, a large bibliographic database of attribute grammars related literature (compiled and available at INRIA) was fed as input to both a Happy parser and a HaGLR parser, and the Happy parser was much more efficient.
Since performance is not the main focus and HaGLR is a generalized parser generator, it is adequate for this work. It outputs a pure parse forest, which is a list of parse trees. This is not the case for all generalized parsers as some performance optimizations change the representation of the parse forests to compact parse forests, which use a different, more compact approach, such that they contain more information but are less intuitive to work around.

3.4 BASIC COMBINATORS

To be able to build complex filters for the task at hand, some basic combinators to build upon are needed. Most of the combinators described in this chapter are defined in the Haskell XML Toolbox library. They enable the creation of filters, as well as manipulation and composition.
The first step is to concretely define a Rose Tree in Haskell. The *NTree* keyword is the data type for Rose Trees defined in the library. A *NTree* of elements of type *a* is defined as a node

(represented by the keyword *NTree*), with an element of type *a*, and a list of *NTree* of the type *a*.

```
data NTree  a   = NTree a [NTree a]
```

It is also important to define a data type for the filters. A filter will, henceforth, be a function that takes a single *NTree* and returns a list of *NTree*, which is empty if the input tree is removed by the filter, or a single element, if the tree is considered correct.

```
type TFilter node = NTree node -> [NTree node]
```

In more detail, the List monad is used just like the Maybe monad, and, while the Maybe monad could be more intuitive for this definition, the end result is almost the same and using the List monad allows for more ease of code writing.

The most important combinator in this work is the combinator *isOf*. Given a predicate on a tree, this combinator produces a filter based on it. This allows for easy filter construction, by just supplying a function that checks a tree and outputs a boolean value.

```
isOf :: (NTree a -> Bool) -> TFilter a
```

The combinator *satisfies* is used to verify if a given tree *satisfies* a filter. That is, if a filter does not remove a tree, then this combinator will output *True*, and if it does, the output will be *False*. This combinator can be useful, together with *isOf*, in converting between predicate and filter types.

```
satisfies :: TFilter a -> NTree a -> Bool
```

To compose filters, the *o* combinator is used. It is purposely named to look like a dot, mimicking the dot from function composition. It is generally used as an infix operator, between the two filters to be composed. This combinator will produce a filter from the two input filters. A tree that satisfies the resulting filter must satisfy both input filters.

```
o  :: TFilter a -> TFilter a -> TFilter a
```

The combinator *orElse* can be used to express alternative between two filters. This combinator takes two filters as input and produces a filter such that, if a tree satisfies either of the input filters, it will satisfy the output filter. It is expected to be used as an infix operator, allowing for easier reading of the code: the alternative between two filters will be read as *filter1 'orElse' filter2*.

```
orElse :: TFilter a -> TFilter a -> TFilter a
```

There are several ways to explore conditional application of filters. However, in this work, only the *when* combinator is used. It will, given two filters, will apply the first if the second is satisfied, or do nothing if it isn't. This combinator behaves like a logical implication, where the second input filter implies the first. It is expected to be used as an infix operator, much like the previous combinators: an implication defined with this filter will be read as *filter 'when' condition*.

```
when :: TFilter a -> TFilter a -> TFilter a
```

The *not* combinator, when applied to a filter, will produce a new filter with a behaviour opposite to the input filter, that is, if the input filter would reject a tree, the output filter will not. It is very useful as some behaviours are best described by negating a filter that does the opposite, as will be seen in future chapters. Note that this combinator is unrelated to the boolean negation operator pre-defined in the Haskell programming language, although it intends to mimick its behaviour.

```
not :: TFilter a -> TFilter a
```

The parse forest produced by the parser is in fact a list of trees, and the filters apply to only one tree. Therefore, to apply a filter to a list of trees, the *$$* combinator is used. It behaves similarly to the pre-defined Haskell function *concatMap*, that is, it applies the filter to all

trees individually and then collects the results, discarding the invalid trees. By using the List monad instead of the Maybe monad, this similarity is more obvious. This combinator is generally used as an infix operator, for example, *filter $$ forest*.

```
($$) :: TFilter a -> [NTree a] -> [NTree a]
```

While the previous combinators were all pre-defined in the Haskell XML Toolbox library, some new ones were also defined to help in building the disambiguation rules that are desired. Since a parse tree is composed of nodes, and each node only contains one chunk of data, a very useful combinator can be defined to check if the root of a tree is equal to the input value. This combinator is the *matches* combinator, and it is one of the main building blocks for the implementation of disambiguation filters described in this work. As an example, *matches "a"* is a filter that will keep a tree with the string *"a"* as the root node, and will discard a tree if it doesn't meet this requirement.

```
matches :: Eq a => a -> TFilter a
matches c = isOf ((c==) . getNode)
```

Since the *matches* combinator, as well as the combinators described before, only work on a local scope, that is, on the root of the tree, there is a need for a way to process the whole tree and gather results. The two following combinators deal with that.
The *every* combinator, given a certain filter that applies to the root node of the tree, will produce a new filter that will apply the input filter to all the nodes of the tree and discard the tree if the input filter fails for any of the nodes. In other words, all nodes of the tree must satisfy the input filter. This combinator is very important, as it is necessary to make sure that all nodes of the tree are correct, and not just the root.

```
every :: TFilter a -> TFilter a
every f = isOf (all (satisfies (every f)) . getChildren) `o` f
```

The *some* combinator is similar to the *every* combinator, but the produced filter is satisfied if at least one node satisfies the filter, as opposed to all nodes. If a property needs to be satisfied in at least one node of the tree, then this combinator is the right option. It is used

less frequently in this work compared to the *every* combinator but it is kept for consistency and variety of tools provided.

```
some :: TFilter a -> TFilter a
some f = f `orElse` isOf (any (satisfies (some f)) . getChildren)
```

3.5 DISAMBIGUATION FILTERS

With these tools, there is more than enough margin to implement the aforementioned disambiguation filters. In the following sections, the types of filters described before are implemented using these combinators. However, it is important to note that they apply to the parse trees produced by the parser, and if a different parser is used, it might be needed to change the filters accordingly.
To build a filter that defines disambiguation rules, it is first needed to take a look at a parse tree and devise an algorithm for checking if it is a valid parse tree. To do so, it is important to understand the structure of the parse trees produced by the parser.
As such, a grammar was produced in a certain way so that it is possible to find all types of ambiguity in it. It is shown in appendix B. All examples will be based on this grammar, showcasing different input strings and what parse trees are produced as a result.

3.5.1 *Associativity Filter*

Given a parser for the grammar described in appendix B, and the simple input string *"1+2+3"*, the output consists of two parse trees, as seen in figure 3. The left one shall be considered the correct interpretation and the right one the undesired interpretation for this example. As such, to write the filter, it is needed to locate what is the pattern that stands out in this example, and then describe a way to remove it using the combinators.
The names of the nodes match with the names of the productions. As an example, the *Exp1* and *Exp2* nodes refer to the *Exp* production. *NTfromT2* refers to a terminal symbol, in this case, the + sign. Finally, *InfixExp2* refers to the *InfixExp* production, which is the one that defines the additions. In this case, the difference between the two parse trees is in whether an *InfixExp2* node is to the left or to the right of their parent *InfixExp2* node. This represents the ambiguity, in which the addition can be left-associative or right-associative. Therefore, the ambiguity can be solved by implementing a filter that removes any tree where there is

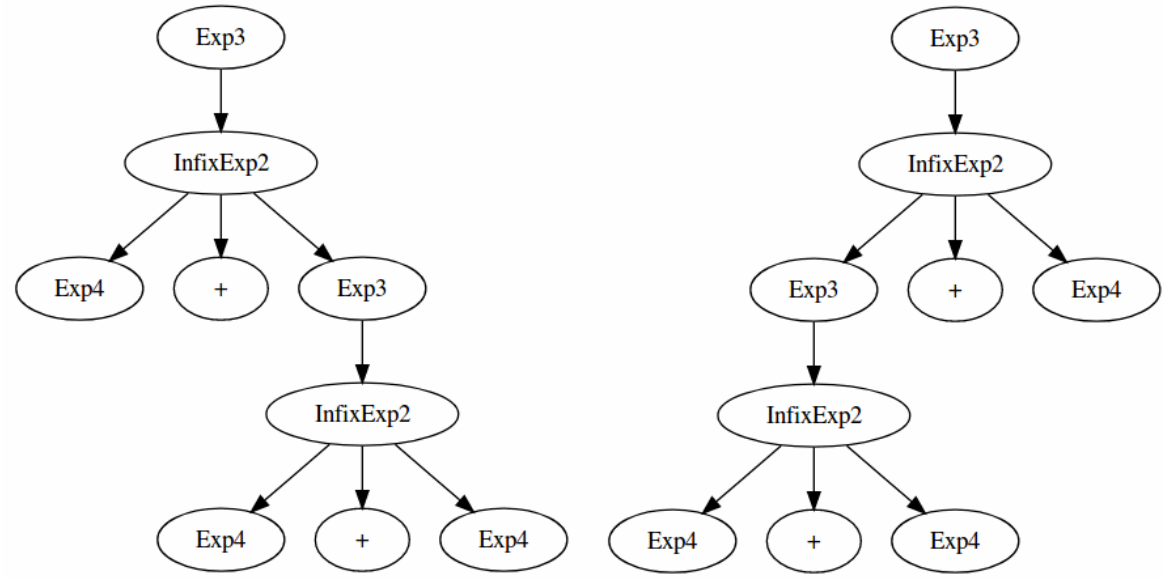


Figure 3: Simplified parse trees for the string "1+2+3;".

an *InfixExp2* node to the right (or left) of another *InfixExp2* node.
The implementation of this filter is rather trivial once the algorithm is defined. This filter will look at the root node, check if it is an *InfixExp2*, and, if it is, check if there is an *InfixExp2* node in the right child of said node. If there is not, then the tree is correct according to the filter.

```
associativity = neg rightNodeCheck `when` matches "InfixExp2"
 where rightNodeCheck = matches "InfixExp2" . head . getChildren . (!!2) .
    getChildren
```

Therefore, the filter is finished and can be read in a reasonably easy way. When the root matches the string *"InfixExp2"*, the rightmost child must not match the string *"InfixExp2"*. Of course, when the root does not match this string, the filter does nothing.
However, this filter does not work as expected on a real parse tree. That is because the filter only applies to the root, and not to the whole tree. In reality, the ambiguity can exist deep into the tree, and so the *every* combinator is needed to apply the filter to all the tree, and discard the tree if any of the nodes fail to satisfy it.

```
disambiguationAssoc = every associativity
```

One last step that can be taken is to generalize the resulting filter. This can be done by having the node name be taken as an argument, and not have it hard-coded. The associativity filter is also generalized to handle associativity to the left and to the right.

```
left_assoc  p = neg (matches p . head . getChildren . (!!2) . getChildren) `
   when` matches p
right_assoc p = neg (matches p . head . getChildren . (!!0) . getChildren) `
   when` matches p

disambiguationAssocGeneric = every (left_assoc "InfixExp2")
```

3.5.2 *Priority Filter*

The priority filter, in its essence, is quite similar to the associativity filter. Given a parser for the grammar described in appendix B, and the simple input string *"1+2*3"*, the output consists of two parse trees, as seen in figure 4. As before, there is a pattern that is undesired and that should be removed in a similar way.

These trees are rather similar to the ones displayed before, and the node naming conventions are the same. In this case, one of the parse trees represents the + symbol being processed before, while the other one represents the * symbol being processed before. To solve this ambiguity, for any node in the tree, if it matches the * symbol, then the children nodes must not match the + symbol. If such nodes exist, they represent a situation where the product happens before the sum, which is not desired behaviour.

```
priority = neg anyChildrenMatches `when` matches "InfixExp0"
 where anyChildrenMatches = (matches "InfixExp5" $$). (concatMap getChildren) .
     getChildren
disambiguationPrio = every priority
```

As before, the *every* combinator is needed to apply the filter to all the tree, and discard the tree if any of the nodes fail to satisfy it. Finally, it is possible to generalize this filter, so as to

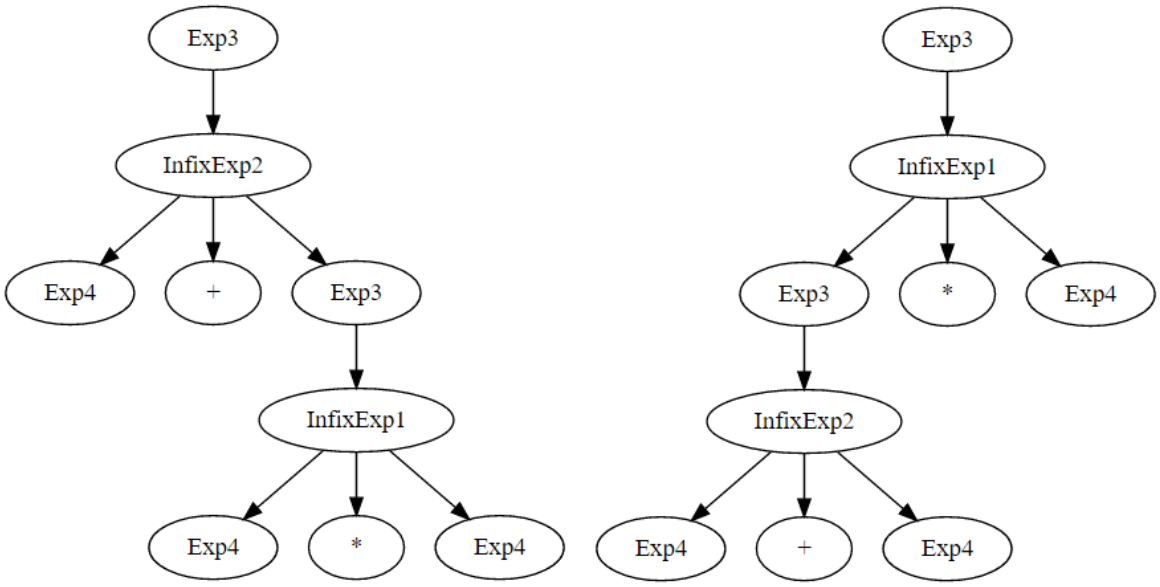


Figure 4: Simplified parse trees for the string "1+2*3;".

allow easier recycling of code.

```
before x y = neg ((matches  x $$). (concatMap getChildren) . getChildren ) ‘
    when‘ matches  y
disambiguationPrioGeneric = every (before "InfixExp5" "InfixExp0")
```

3.5.3 *Reject Filter*

Given a parser for the grammar described in appendix B, and the simple input string *"x = true"*, the output consists of two parse trees, as seen in figure 5. By looking at them, it is possible to understand that the parser can interpret the string *true* as either a boolean value, that is, a *"Bool"* node, or an identifier, that is, an *"Id"* node. While both are technically correct, this problem needs to be solved by rejecting certain keywords as identifiers, thus the need for reject filters.

Since this parser is a scannerless parser, a string is actually a sequence of tokens, where each token is a character. To recover the string from the sequence of tokens, the *implodeSubTree* function is used. It is part of the tools available in the HaGLR parser, and it converts a sequence of tokens into a string. The resulting string is reversed, and thus the *reverse* function is used to fix the strings and enable adequate comparison between them.

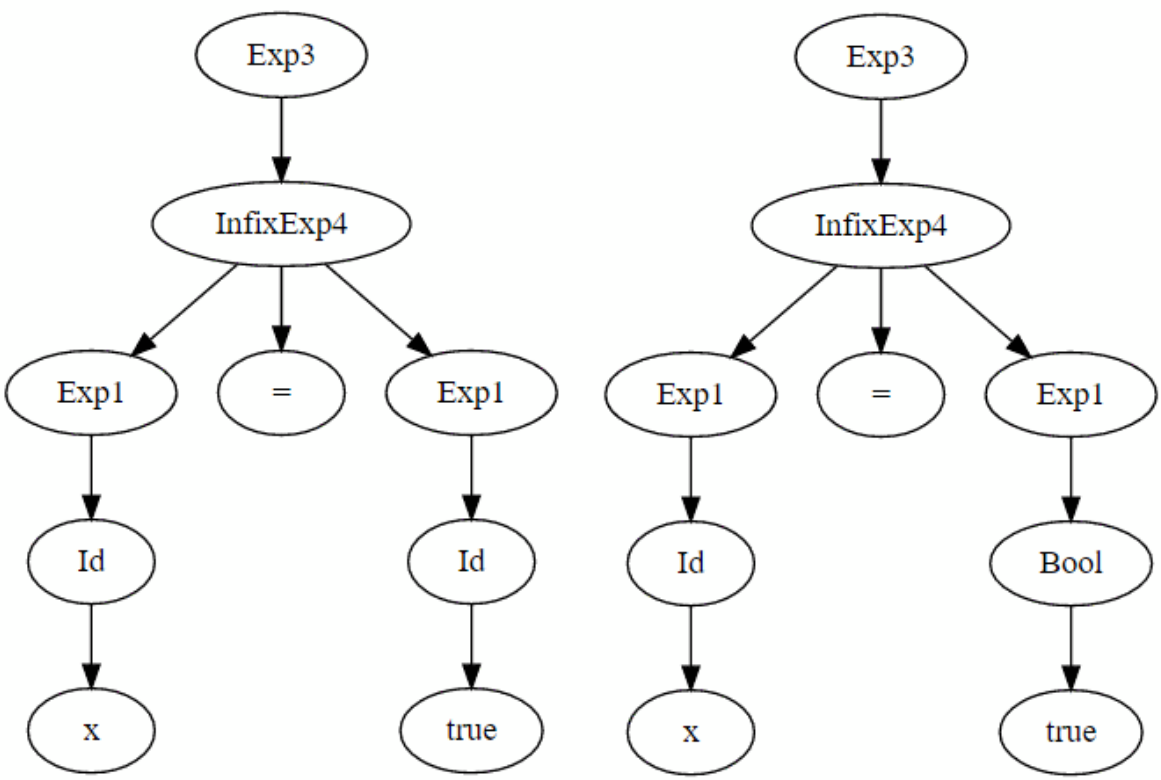


Figure 5: Simplified parse trees for the string "x = true;".

According to the definition of this filter, when a node matches the *"Id"* production, then the string it derives into cannot match any of the desired keywords. The following implementation only considers the keywords *"true"* and *"false"* but it is simple to expand to any more desired keywords.

```
reject = neg stringMatchesKeywords `when` matches "Id"
 where stringMatchesKeywords = (matches (reverse "true") `orElse` matches (
     reverse "false")) . head . getChildren . implodeSubTree
disambiguationRej = every reject
```

As before, the *every* combinator is used to apply this filter to the whole tree. Of course, only the *"Id"* nodes can be affected by it, but they can be located anywhere on the tree, hence why this combinator is needed. One last step is to generalize this filter.

```
reject    w p = neg (matches       (reverse w) . head . getChildren .
    implodeSubTree) `when` matches p
disambiguationRejGeneric = every (reject "true" "Id" `o` reject "false" "Id")
```

3.5.4 *Follow Filter*

Given a parser for the grammar described in appendix B, and the simple input string *"int x [] = [12 3]"*, the output consists of two parse trees, as seen in figure 6. Since whitespace is not directly specified in the grammar, if there are not any other separators, the parser cannot distinguish whether the string *"12"* is one or two values. As such, the follow filters are used to solve this ambiguity.

In this situation, it is desired that the *"Values1"* production has the longest possible match in each of its children, that is, tries to incorporate as many characters as possible into each children. In this situation, it is enough to specify that either the left child ends with a character that is not a number, or the right child starts with a value that is not a number. This forces the output to only contain this production when there are two values separated by a whitespace in the input, because if there are no whitespaces, the filter will reject the whole tree.

The implementation of this filter just specifies that, when the root node is a *"Values1"*

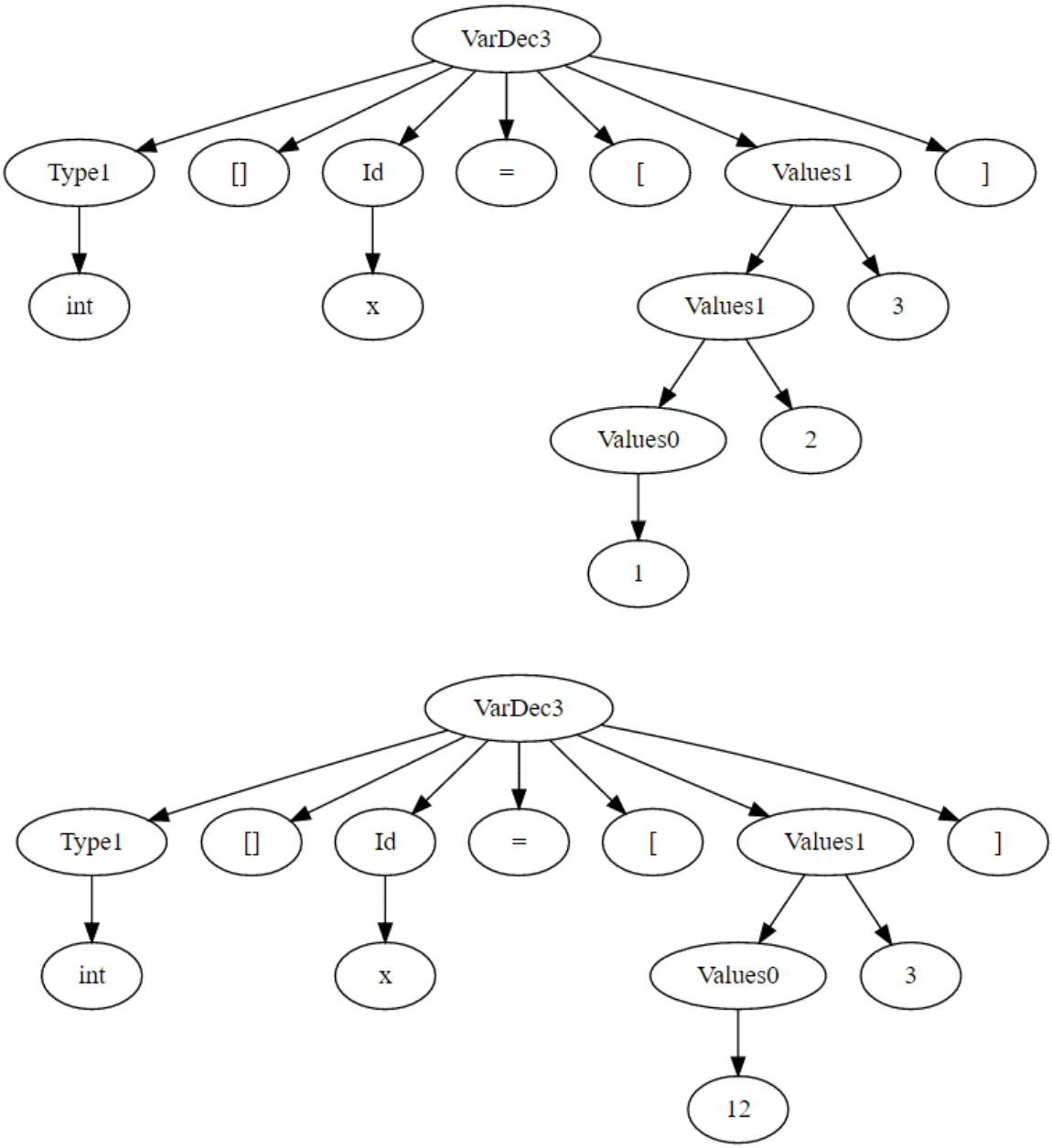


Figure 6: Simplified parse trees for the string "int [] x = [12 3];".

node, then the first child's digested string must end with a desired character or the second child's digested string must start with one, so that between the two, there is at least one whitespace. It is important to note that the parser assembles whitespaces into various nodes automatically, and it is taken advantage of this fact to describe this filter.

```
follow = (firstEndsGood `orElse` secondStartsGood) `when` matches "Values1"
 where firstEndsGood =    isOf (not . isDigit . head . getLastTerm . (!!0) .
    getChildren)
       secondStartsGood = isOf (not . isDigit . head . getFirstTerm . (!!1) .
           getChildren)
disambiguationFol = every follow
```

As before, the *every* combinator is used to apply this filter to the whole tree. Of course, only the *"Values1"* nodes can be affected by it, but they can be located anywhere on the tree, hence why this combinator is needed. One last step is to generalize this filter.

```
follow    t r = (firstEndsGood `orElse` secondStartsGood) `when` matches t
 where firstEndsGood    = isOf $ flip notElem r . head . getLastTerm  . (!!0) .
     getChildren
       secondStartsGood = isOf $ flip notElem r . head . getFirstTerm . (!!1) .
            getChildren
disambiguationFolGeneric = every (follow "Values1" "123456789")
```

3.5.5 *Preference Filter*

Given a parser for the grammar described in appendix B, and the simple input string *"if true then if false then 1 else 2"*, the output consists of two parse trees, as seen in figure 7. This is a rather famous ambiguity problem generally described as the *dangling else* problem. In this input string, there are two *if...then* clauses, and one *else* keyword, but there is no clue as to which *if* statement does it belong to. Therefore, the parser will generate two interpretations, in which the *else* keyword will associate with either of the *if...then* clauses.

In this situation, both interpretations are correct, but the developer will want to prefer one over the other. This is where the preference filter comes in. It will only act on ambiguities where several interpretations are correct but there is one that is desired over the others.

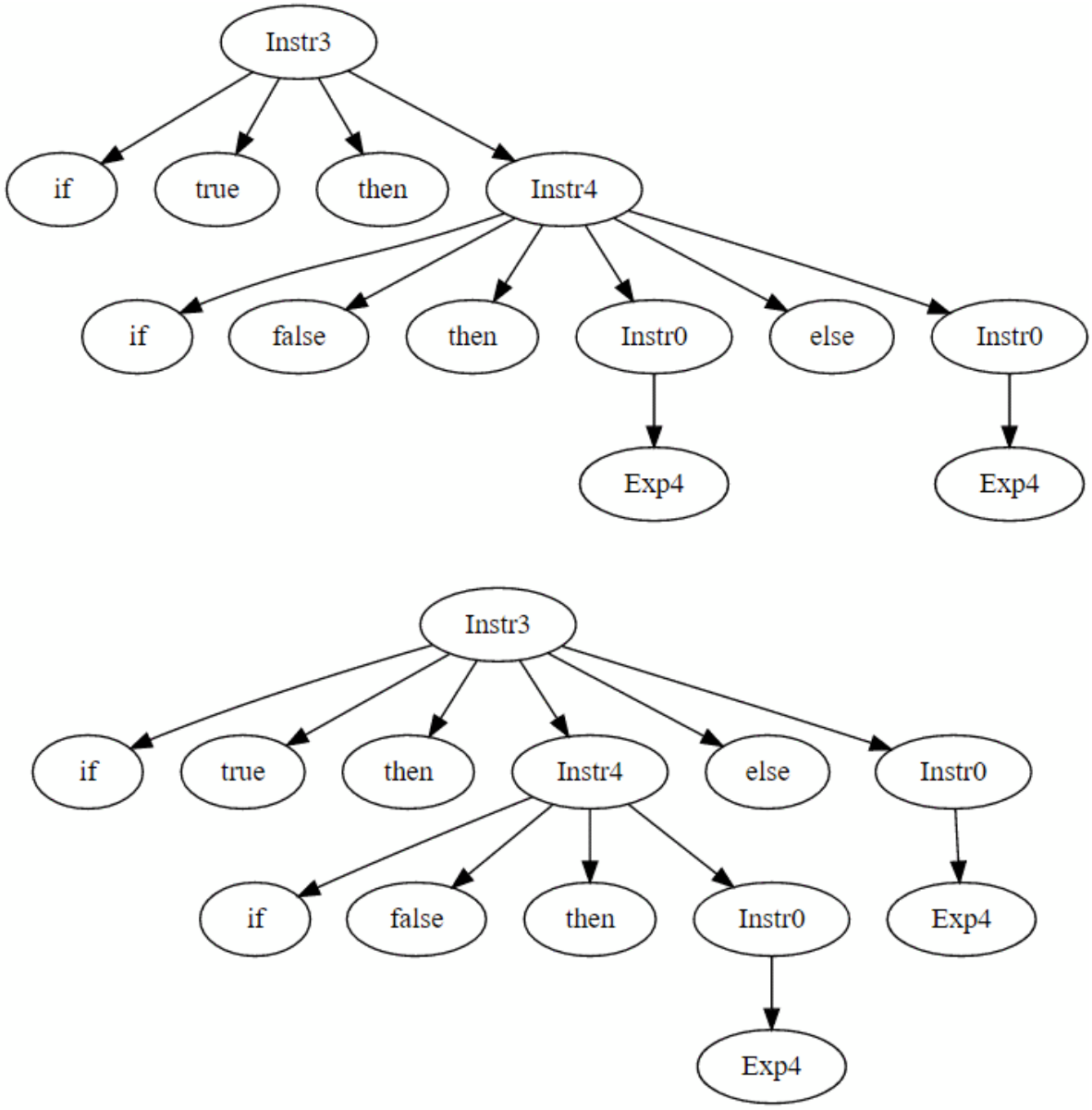


Figure 7: Simplified parse trees for the string "if true then if false then 1 else 2;".

This filter, however, does not behave similarly to the other filters. It works by comparing different parse trees, and then choosing the best one, which is fundamentally different to the other filters which operate on a single tree. Therefore, the implementation of this filter is also very different, and is just a function which operates on lists. This function compares each parse tree to every other tree, and discards a parse tree if it is considered less interesting than any other one.

```
preference bad_token good_token l = preference' l l
 where preference' [] l = []
       preference' (x:xs) l = if any (x `isWorseThan` ) l then preference' xs l
            else x : preference' xs l
       isWorseThan (NTree x t) (NTree y tt) = if x == y then or (zipWith
           isWorseThan t tt) else x == bad_token && y == good_token
disambiguationPrefGeneric = preference "Instr3" "Instr4"
```

3.5.6 *Arbitrary Filters*

While filter combinators can be used to implement already known disambiguation rules, they can also be used to implement new concepts and ideas that generally are not possible to implement in the disambiguation rules of most parsers. This allows the developer to express any desired rules, without the limitations of not being able to fine-tune the filters. As an example, in this chapter, it will be presented a filter that associates any sum operations to the left, until an *if* clause is found, and inside it, the sum operations will associate to the right. While there is no immediate use for this filter, it is a good example of different behaviour implemented into a filter.
To begin with, it is necessary to explore one more combinator from the combinator library. It would be possible to express this filter with the combinators shown before but it is simpler to just use the simplest tools. This combinator is named *iff* and is similar to the *when* combinator. It accepts three arguments, which correspond to *if*, *then* and *else* clauses.

```
ff = iff (matches "Instr3") rightAssocEverything leftAssocUntil
 where rightAssocEverything = every (right_assoc "InfixExp2")
       leftAssocUntil = isOf (all (satisfies ff) . getChildren) `o` left_assoc
           "InfixExp2"
```

The *iff* combinator is fed three arguments, where the first is just to check if the current node matches the *if* instruction. If so, the *rightAssocEverything* portion of the code is run, which just applies the *right_assoc* combinator shown before to all the subtrees from that point onwards. If the matching fails, as seen in the *leftAssocUntil* portion of the code, the *left_assoc* combinator is applied to the current node, and the whole filter is recursively applied to all the subtrees.

4

IMPLEMENTATION DETAILS

In this chapter, it will be shown how to generate a parser and link the disambiguation filters to it. Some practical results are described, as well as a comparison with the previous implementation of the HaGLR parser generator. However, it is difficult to compare the obtained results, partially because of earlier versions of the parser generator not being able to produce a parser for examples that are trivial for the current implementation.

4.1 USAGE EXAMPLE

To generate a parser, a grammar is needed. In this case, the grammar described in appendix B will be used, as it is the grammar that has been referenced throughout this work, and it will be stored in a file named *Example-filters.txt*.
The first step is to run the parser generator. There are several ways of doing this, as it can be run from the interpreter, it can be run with the command *runhaskell*, from the makefile or it can be compiled and run as an executable. The following example is the *runhaskell* method, which is invoked when the makefile command *make example-filters* is run. This command invokes the main function in the *Tablegenerator* module, which contains the tools to generate a parser. The following arguments specify the input grammar and the starting production, which will be *"Instrs"*, which is a list of instructions.

```
runhaskell Language/HaGLR/TableGenerator.hs
           -g "Examples/Example-filters.txt" -s "Instrs"
```

This command will generate several files, which contain all the relevant information for the parser, such as the action and lookup tables. The *GLRActionsTable* module contains the main functions that perform the actual parsing. It is important to note the function *glr_parser_aterm*, as this function takes a string as input and outputs a parse forest, according to the grammar used to generate the parser.

Now, an input string is needed. To display all the disambiguation rules presented before, a bigger input string is used. The following input is a toy program, that runs through an array and checks if it contains an even number.

```
int [] input = [ 11 22 33 4+73+2 ];
bool found = false;
int counter = 0;
while (counter < 4){
    if found == false
        then if input[counter] % 2 == 0
                then found = true;
            else break;
}
```

It contains all the ambiguities presented before:

- Follow - some digit sequences inside the array declaration can be interpreted by the parser as either one number or two numbers.
- Associativity - the last value in the array is a sum of three numbers, which can be associative to the left or right.
- Reject - the keywords *true*, *false* and *break* can be interpreted as identifiers.
- Priority - the expression *"input[counter] % 2 == 0"* is ambiguous as there is no set priority for the equality and module operations.
- Preference - there is a dangling *else* keyword that is not immediately associated to either of the *if* clauses, resulting in ambiguity.

To incorporate disambiguation rules using filter combinators, the filters are defined in a *Disambiguation* module. The filters described before are implemented in this module. From here, it is possible to use some or all of the filters, and test the results. The following is a demonstration of the number of parse trees generated by the parser, and then the resulting

```
*GLRActionsTable> s <- readFile "Examples/input-filters.txt"
*GLRActionsTable> let p = glr_parser_aterm s
*GLRActionsTable> length p
1536
*GLRActionsTable> length $ every (f_assoc_plus) $$ p
1024
*GLRActionsTable> length $ every (f_assoc_plus `o` f_prio) $$ p
512
*GLRActionsTable> length $ every (f_assoc_plus `o` f_prio `o` f_reject ) $$ p
32
*GLRActionsTable> length $ every (f_assoc_plus `o` f_prio `o` f_reject `o` f_follow ) $$ p
2
*GLRActionsTable> length $ f_pref $ every (f_assoc_plus `o` f_prio `o` f_reject `o` f_follow ) $$ p
1
*GLRActionsTable> length $ f_disamb p
1
*GLRActionsTable>
```

Figure 8: Testing number of parse trees before and after disambiguation

number of parse trees after performing disambiguation. The input program is stored in s, the parse forest in the list of trees p and then it is measured the length of the list p before and after disambiguation, that is, the number of parse trees in each step.

4.2 PERFORMANCE ANALYSIS

The initial implementation of the HaGLR parser generator was intended for educational purposes and was extremely inefficient, and some optimizations were applied so as to enable the use of this parser generator for real-world cases, even if still inefficiently so. These optimizations do not change the algorithms used in the code, only the way the data is manipulated. Some examples of optimization include:

- Representation of matrices as a vector of vectors instead of a list of lists, as a vector of vectors allows for $O(1)$ operations while a list of lists only allows $O(n^2)$ operations.
- Removal of duplicate computations hidden in the code that were causing big increases in runtime.
- Replacement of the *String* data type and operations with the *ByteString* data type, as the latter is more efficient for writing big chunks of text, such as the files generated by the parser generator.
- Factorization of some well-defined data that is constantly repeated in the generated lookup tables for the parser, such that the data is replaced by integer values, which are indexes in a separate array of data. This results in much smaller tables, allowing for faster compilation of the generated parser and with less memory usage.

Some benchmarks were performed to compare the results from the initial and current implementations, as well as the impact of the disambiguation filters in terms of performance. All the benchmarks present in this work were run 10 times, from which the largest and smallest value were removed, and then the rest of them averaged. This is to ensure that the benchmark is not influenced by spikes in CPU usage or any other external factors. The benchmarks were run on a system described in table 1.

Three grammars were used in the benchmarking process. The simplest is the *Small* grammar,

Processor	Intel® Core™ i5-7200U Dual-Core, 2.50 GHz, 3MB cache
Hard Disk	SSD 256 GB PCIe® NVMe™ M.2
RAM	8GB SDRAM LPDDR3-1866
OS	Ubuntu 16.04 LTS

Table 1: Specifications of the hardware used for the tests.

described in appendix A, and it consists of simple arithmetic expressions and declaration of variables. The *Filters* grammar, present in previous examples throughout this work, described in appendix B, contains several possible expressions and ambiguities, behaving as a toy programming language. The *Tiger* grammar, described in appendix C, can be used to generate a parser for the Tiger programming language and is a real-world example that is used to show how this work behaves when presented with a bigger grammar.

The table 2 displays the results of measuring the generation time of the parsers for the *Small*, *Filters* and *Tiger* grammars, using both the initial and current implementations of the parser generator. The generation time of the parsers is arguably not important as the parser is expected to only be generated once, and the filters are independent. However, during development, there can be a need to generate the parser various times, for example, if the grammar is incomplete or has errors that need to be fixed, and as such, having a low parser generation time is helpful.

	Small	**Filters**	**Tiger**
Initial	1.224	20.132	860.340
Current	0.273	4.571	288.594

Table 2: Parser generation benchmarking, in seconds.

The table 3 shows the time, in seconds, that is spent in compiling each of the previously generated parsers. This compilation is done using the *Glasgow Haskell Compiler (GHC)*, with the optimization flag and forcing recompilation, so that no files are re-used between tests. The *Tiger* parser generated by the initial implementation of the parser generator, after around 5 hours, crashed due to running out of memory. The parsers generated by the

current implementation of the parser generator are faster to compile due to some memory optimizations applied to tables used in them.

	Small	**Filters**	**Tiger**
Initial	18.729	1067.954	–
Current	10.867	45.413	189.538

Table 3: Parser compilation benchmarking, in seconds.

The parser generated by the initial implementation of the parser generator was incredibly inefficient, and the benchmark present in the table 4 is a demonstration of that. Using only the parser for the *Small* grammar, strings containing the sum of the number 1 various times were fed as input. Each column refers to the number of different interpretations of such inputs.

	4862	**16786**	**57876**
Initial	3.044	12.438	51.534
Current	0.178	0.567	1.950

Table 4: Parser usage, in seconds. Note that the columns refer to the number of ambiguous outputs.

As a final benchmark, for the *Filters* and *Tiger* languages, the ambiguous input shown in listing 4.1 and the solution to the n-Queens problem shown in appendix D were used, respectively. They were chosen as the first one is known to be ambiguous, since it was built for such purposes, and the last one is a real-world example. For each of those, it was measured, in seconds, the time it takes to parse without and with disambiguation filters. For the *Filters* input, the use of disambiguation filters actually reduces the time it takes to parse the input. This happens due to the laziness of Haskell - while the parsing and disambiguation steps, in theory, are separated, the actual result is that they intertwine and therefore unnecessary computations are thrown out by the filters. However, for the *Tiger* input, which is more complex but less ambiguous, the filters actually increase the execution time, as they add more computational weight to the process and do not benefit from lazy computations as much.

	Filters	**Tiger**
No filters	0.436	2.119
With filters	0.329	2.829

Table 5: Comparison on the runtime without and with filters, in seconds, for the current parser implementation.

5

CONCLUSIONS

This thesis presents an alternative for parser development with several advantages and disadvantages. The process of building a parser using a parser generator is generally very straightforward, but also very limited, at least in terms of disambiguation, and the only tool available for disambiguation is the one provided by the parser generator, which modifies the parser. While this is optimal in terms of performance, there are also several shortcomings, such as lack of power for the developer to define different rules not contemplated by the parser generator, or the inability to change the disambiguation rules after the parser is generated, resulting in the need to generate a new parser.
With this tool, it is possible to fine-tune the disambiguation process as much as needed, by building whichever rules are needed with no limitations. The use of combinators to build these disambiguation rules as filters allows for clean, simple and composable code that can be easy to write and maintain. While it can be complicated to understand how to write a custom disambiguation filter for the developer if they have no experience on the matter, there are some pre-defined filters that allow for the well-known disambiguation rules to be implemented without the need for implementing new behaviours.

5.1 FUTURE WORK

There are various ways this work can be expanded in the future. As this is a tool different from the tools generally used for the development of parsers, there are also various ways to evolve this work into a more powerful tool, be it either by increasing its expressive power, performance or ease of use. Some possible ways to expand on this work include:

- Develop a graphical interface for this tool. This tool would enable the developer to just draw the filters as blocks, chain them as needed, and then convert the resulting model into either Haskell code or a functional Haskell module, to be imported onto the parser code. An interesting alternative would be for the graphical interface to show the developer a parse tree and allow for the developer to describe the rule by

interacting with the tree in certain ways, such as by selecting two adjacent nodes and selecting an option that forces them not to appear together.

- Study the influence of the order of application of the filters. Since the filters do not modify the parse trees, the order in which they are applied does not influence the result, but it can influence the execution time of the filters. If several filters are applied and they do not modify the parse forest, they are just a waste of computational resources, and if there is one filter that reduced the parse forest into a single parse tree in the list of filters to be applied, if said filter is applied first, the rest of the filters only have to be applied to one parse tree as opposed to a whole parse forest. As such, it is relevant to study ways to optimize the order of filter application, as it can reduce the performance burden of the disambiguation filters.

- Develop a tool that can, based on the disambiguation filters presented in this work, actually modify a parser to obey the rules they describe. Alternatively, a tool can be developed to generate disambiguation filters based on the disambiguation rules as they are described in the most popular parsers, allowing developers to start using this tool without having almost any knowledge on disambiguation filters.

Appendices

A

SMALL GRAMMAR

```
#CFG

Exps -> Exp

Exp -> InfixExp
    |  Integer
    |  VarDec
    |  Identifier Exp


InfixExp -> Exp '*' Exp
         |  Exp '+' Exp
         |  Exp '&' Exp
         |  Exp '|' Exp
         |  Exp '=' Exp
         |  Exp "<>" Exp
         |  Exp '>' Exp

VarDec -> "int" '*' Exp ';'

#RegExp

Integer    -> [0-9]+
Identifier -> [a-c]+
Ws -> (' '|'\n'|'\r'|'\t'|'\v')*
```

B

FILTERS GRAMMAR

```
#CFG
Instrs -> {Instr}+

Exp -> InfixExp
    |  Integer
    |  Id
    |  Bool
    |  Id '[' Exp ']'

Instr -> Exp ';'
        |  "if" Exp "then" Instr "else" Instr
        |  "if" Exp "then" Instr
        |  "while" '(' Exp ')' '{' {Instr}* '}'
        |  VarDec
        | "break" ';'

Bool -> "true"
    | "false"

InfixExp -> Exp '*' Exp
         |  Exp '+' Exp
         |  Exp '%' Exp
         |  Exp '=' Exp
         |  Exp "==" Exp
         |  Exp '<' Exp
         |  Exp '>' Exp

VarDec -> Type Id '=' Exp ';'
        | Type Id ';'
```

```
        | Type '[' ']' Id ';'
        | Type '[' ']' Id '=' '[' Values ']' ';'

Type -> "int"
      | "bool"

Values -> Exp
        | Values Exp

#RegExp
Id -> [a-z]+
--Chars  -> '"'[^\"]+'"'
Integer    -> [0-9]+
Ws -> (' '|'\n'|'\r'|'\t'|'\v')*
```

C

TIGER GRAMMAR

```
#CFG

Exp -> LValue
     | "break"
     | "nil"
     | Integer
     | StringLit
     | SeqExp
     | Negation
     | CallExp
     | InfixExp
     | ArrExp
     | Assignment
     | WhileExp
     | ForExp
     | IfThen
     | IfThenElse
     | RecExp
     | LetExp

Decs  -> { Dec }*

Dec -> TyDec
     | VarDec
     | FunDec

TyDec -> "type" Identifier '=' Ty

Ty -> Identifier
```

```
    | ArrTy
    | RecTy

ArrTy -> "array" "of" Identifier


RecTy -> '{' '}'
       | '{' FieldDecs '}'

FieldDecs -> FieldDec { ',' FieldDec }*

FieldDec  -> Identifier ':' Identifier

FunDec -> "function" Identifier '(' FieldDecs ')'               '=' Exp
        | "function" Identifier '('           ')'               '=' Exp
        | "function" Identifier '(' FieldDecs ')' ':' Identifier '=' Exp
        | "function" Identifier '('           ')' ':' Identifier '=' Exp


VarDec -> "var" Identifier               ":=" Exp
        | "var" Identifier ':' Identifier ":=" Exp


LValue -> Identifier
        | OtherLValue

OtherLValue -> Identifier '[' Exp ']'
             | OtherLValue '[' Exp ']'
             | LValue '.' Identifier


SeqExp -> '(' ')'
        | '(' ExpSeq ')'

ExpSeq  -> Exp { ';' Exp }*

ExpList -> Exp ',' ExpList
         | Exp
```

```
Negation -> ’-’ Exp

CallExp -> Identifier ’(’ ExpList ’)’
         | Identifier ’(’ ’)’

InfixExp -> Exp ’*’  Exp
          | Exp ’/’  Exp
          | Exp ’+’  Exp
          | Exp ’-’  Exp
          | Exp ’=’  Exp
          | Exp "<>" Exp
          | Exp ’>’  Exp
          | Exp ’<’  Exp
          | Exp ">=" Exp
          | Exp "<=" Exp
          | Exp ’&’  Exp
          | Exp ’|’  Exp


ArrExp -> Identifier ’[’ Exp ’]’ "of" Exp

RecExp -> Identifier ’{’ ’}’
        | Identifier ’{’ FieldCreates ’}’

FieldCreates -> FieldCreate { ’,’ FieldCreate }*

FieldCreate  -> Identifier ’=’ Exp

Assignment -> LValue ":=" Exp

IfThenElse -> "if" Exp "then" Exp "else" Exp

IfThen     -> "if" Exp "then" Exp

WhileExp   -> "while" Exp "do" Exp

ForExp     -> "for" Identifier ":=" Exp "to" Exp "do" Exp
```

```
LetExp     -> "let" Decs "in" ExpSeq "end"

StringLit -> '"' StrLitAux '"'

#RegExp

Integer    -> [0-9]+

Identifier -> [a-zA-Z][a-zA-Z0-9]*

StrLitAux  -> [^\"]+

Ws -> (' '|'\n'|'\r'|'\t'|'\v')*
```

D

8QUEEN SOLUTION IN TIGER

```
let
    var N := 8

    type intArray = array of int

    var row := intArray [ N ] of 0
    var col := intArray [ N ] of 0
    var diag1 := intArray [N+N-1] of 0
    var diag2 := intArray [N+N-1] of 0

    function printboard() =
       (for i := 0 to N-1
     do (for j := 0 to N-1
          do print(if col[i]=j then "O" else ".");
         print("\n"));
         print("\n"))

    function try(c:int) =
(
     if c=N
     then printboard()
     else for r := 0 to N-1
       do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
               then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
                 col[c]:=r;
                     try(c+1);
             row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)

)
```

```
 in try(0)
end
```

BIBLIOGRAPHY

Ali Afroozeh and Anastasia Izmaylova. *Faster, Practical GLL Parsing*, pages 89–108. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-46663-6. doi: 10.1007/978-3-662-46663-6_5. URL https://doi.org/10.1007/978-3-662-46663-6_5.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*, pages 125–131, 1959. URL http://dblp.uni-trier.de/db/conf/ifip/ifip1959.html#Backus59.

F. L. Deremer. Practical translators for lr(k) languages. Technical report, Cambridge, MA, USA, 1969.

João Fernandes, João Saraiva, and Joost Visser. *Generalised LR Parsing in Haskell*. 2004.

Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. 1979.

Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. *Generalised Parsing: Some Costs*. 03 2004.

Simon Marlow and Andy Gil. *Happy User Guide*. 2001.

Terence Parr and Kathleen Fisher. Ll(*): The foundation of the antlr parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993548. URL http://doi.acm.org/10.1145/1993498.1993548.

Alan J. Perlis, Mary Shaw, and Frederick Sayward, editors. *Software Metrics: An Analysis and Evaluation*. MIT Press, Cambridge, MA, USA, 1981. ISBN 0262160838.

Uwe Schmidt, Martin Schmidt, and Torben Kuseler. hxt: A collection of tools for processing xml with haskell. https://github.com/UweSchmidt/hxt, 2016.

Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985. ISBN 0898382025.